MPICH-G2 Collective Operations

Performance evaluation, optimizations

Sébastien Lacour
lacour@mcs.anl.gov

Supervisors: **Nick Karonis, Ian Foster**
{karonis,foster}@mcs.anl.gov

Argonne National Laboratory
Mathematics & Computer Science Division
June – September 2001

Internship for the academic year of "Maîtrise" in Computer Science at the University "École Normale Supérieure de Lyon", France.

Abstract (in French)

Pendant ce stage à Argonne, j'ai travaillé sur les opérations collectives de MPICH-G2, l'implémentation de MPI à Argonne Nat'l Laboratory, configurée avec le support de communications Globus-2 : c'est le seul support qui permette de faire du *metacomputing*, car il parle plusieurs protocoles différents.

La première partie du stage a consisté à implémenter une méthode *fiable* et *reproductible* de mesure de performance de la fonction `MPI_Bcast`, et à intégrer ce code dans le logiciel "perftest" d'évaluation de performance distribué avec MPICH.

Ensuite, j'ai enrichi MPICH-G2 de la connaissance de la topologie des processeurs sur lesquels s'exécutent les processus MPI.

La troisième partie du travail a consisté à ré-écrire les 5 opérations collectives les plus utilisées pour les optimiser : ma nouvelle implémentation tient compte de la topologie sous-jacente des processeurs et des liens de communication qui les relient, afin par exemple de limiter les communications intercontinentales via TCP qui présentent une insupportable latence.

Enfin, j'ai rendu cette information sur la topologie des processeurs accessible (de façon sécurisée) au niveau utilisateur, de sorte que le programmeur puisse en tirer parti pour optimiser ses applications MPI.

Contents

During my stay in Mathematics & Computer Science Division at Argonne National Laboratory, I focused on the collective operations in MPICH-G2[1], a particular communication "device" (*Globus-2*) of Argonne's implementation of MPI[2]. That "device" relies on Globus[3] to offer a *grid-enabled* implementation of MPI and constitutes a real step towards *metacomputing* on heterogeneous distributed computing systems [3].

1 Performance evaluation of `MPI_Bcast`

The *BroadCast* operation[4] is one of the most widely used collective operations of MPI. As a user writes a parallel program using the Message Passing paradigm, he needs to determine what MPI library will be the most efficient for his application. Thus accurate measurements of `MPI_Bcast` performance are required.

In the very first part of my work, I implemented an already existing methodology [2] to time `MPI_Bcast` in a *fair* and *reliable* way.

1.1 The difficulties in timing `MPI_Bcast`

To obtain reproducible results, it is obvious that we cannot time only *one* `MPI_Bcast`: we need to measure the time it takes to perform several `MPI_Bcast` operations and divide the total time by the number of `MPI_Bcast`s posted.

When posting several `MPI_Bcast` repeatedly, the broadcast messages proceed through the network concurrently: that effect is called *pipelining effect* [2] as illustrated on figure 1. That is why the algorithm shown on figure 2 is not correct: it gives only a *lower bound* for the time it takes to broadcast.

Posting an `MPI_Barrier` between each `MPI_Bcast` eliminates the pipelining effect preventing two `MPI_Bcast`s from being in progress concurrently (see figure 3). But we would then need to evaluate the time it takes to perform an `MPI_Barrier` facing the same problems as for `MPI_Bcast`. Note that the `MPI_Barrier`s can also proceed concurrently with the preceding and following `MPI_Bcast`s (see figure 4).

Several other difficulties are presented in the article [2].

[1]see: `http://www.globus.org/mpi`.

[2]Message Passing Interface, see: `http://www.mcs.anl.gov/mpi`.

[3]see: `http://www.globus.org`.

[4]`MPI_Bcast`: one so called "root" process sends a piece of data to *all* the other processes in the same set; for more details, see [4].

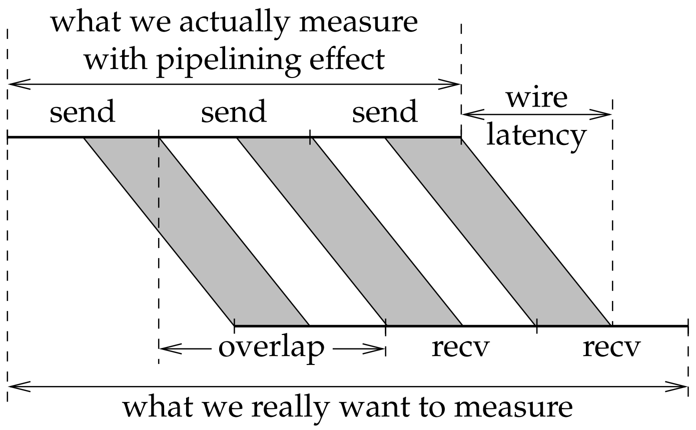

what we actually measure with pipelining effect
send
send
send
wire latency
overlap
recv
recv
what we really want to measure

Figure 1: Consecutive `MPI_Bcast`s proceed concurrently through the network.

Root process:

```
t_0  = get time
for n=1 to MAX
   MPI_Bcast
t_1  = get time
time_to_bcast = $\frac{t_1 - t_0}{MAX}$
```

All other processes:

```
for n=1 to MAX
   MPI_Bcast
```

Figure 2: A too simple timing algorithm.

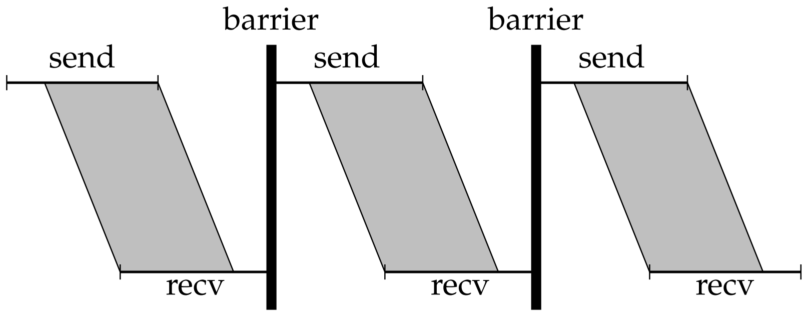


Figure 3: Inserting `MPI_Barrier`s between the `MPI_Bcast`s eliminates the pipelining effect.

1.2 The new timing methodology

Before presenting the new methodology of the article [2], we need to introduce the *LogP* model [1]. As shown on figure 5:

- the time it takes a process to send a message is the *send overhead* o_s,
- the time it takes a process to receive a message is the *receive overhead* o_r,

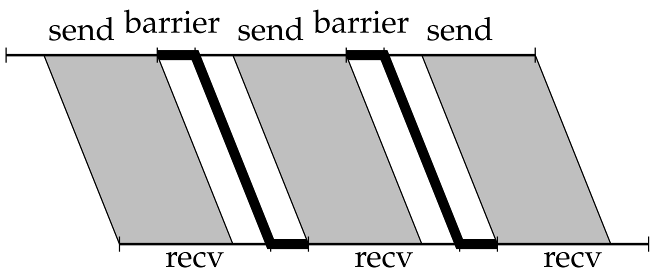


Figure 4: Inserting `MPI_Barrier`s between the `MPI_Bcast`s is not a good solution.

- the amount of time it takes a byte to transit from its source to its destination is the *wire latency* L,
- the minimum interval between consecutive sends/receives is the gap g.

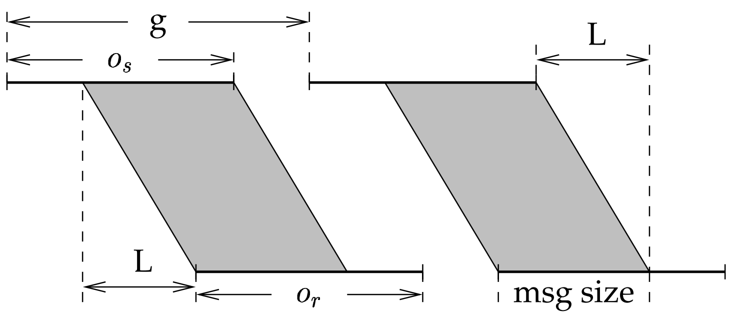


Figure 5: Relevant parameters of the *LogP* model.

The *LogP* model is both simple and complete enough to be much more realistic than the *PRAM* model. For simplicity, we consider $(o_s \simeq o_r) = o$.

Figure 6 shows how the different processes involved in an `MPI_Bcast` receive and relay the message using a binomial-tree algorithm (root process = 0). We need to measure the *operation latency* $OL = t_1 - t_0$. The idea consists in measuring the operation latencies OL_i for each process i, and $OL = max_i(OL_i)$.

Figure 7 reproduces the algorithm used to measure OL_i as found in the article [2]. This algorithm is repeated for every process i which sends an acknowledgement (*ack*) to the root process after completing each `MPI_Bcast`. The root does not start another `MPI_Bcast` until it receives this *ack*. The overhead to send/receive the acknowledgement is corrected very easily measuring the latency of a single point-to-point message from process i to the root.

The extra `MPI_Barrier` and `MPI_Bcast` before the actual measurement is used to "prime the line": the first communication might take more time than the following com-

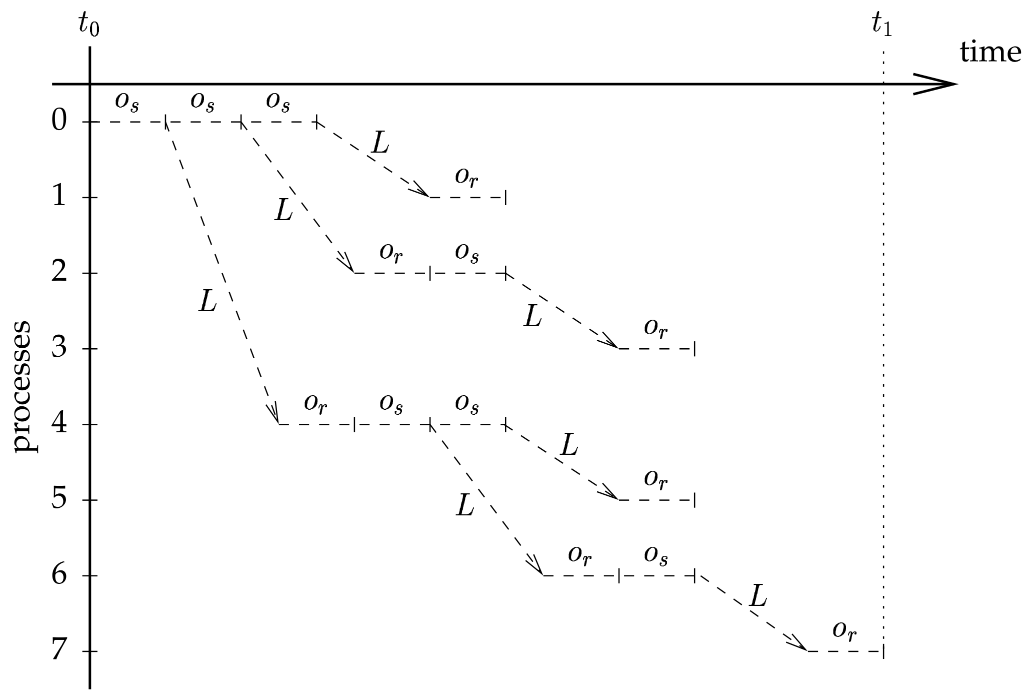


Figure 6: Binomial-tree algorithm for `MPI_Bcast` from root = p_0.

Root process:

```
    t₀ = get time
    for n=1 to MAX
        MPI_Send ack to i
        MPI_Recv ack from i
    t₁ = get time
    ack_time_i = (t₁ - t₀) / (2 * MAX)

    MPI_Barrier
    MPI_Bcast
    MPI_Recv ack from i

    t₀ = get time
    for n=1 to MAX
        MPI_Bcast
        MPI_Recv ack from i
    t₁ = get time
    OL_i = (t₁ - t₀) / MAX - ack_time_i
```

Process i:

```
    for n=1 to MAX
        MPI_Recv ack from root
        MPI_Send ack to root

    MPI_Barrier
    for n=1 to MAX+1
        MPI_Bcast
        MPI_Send ack to root
```

All other processes:

```
    for n=1 to MAX+1
        MPI_Bcast
```

Figure 7: The new methodology to time `MPI_Bcast`: fair and reliable.

munications. There might be some acknowledgements which arrive to the root before the latter completes the `MPI_Bcast`: that does not matter because such an *ack* is certainly not coming from the process with the largest OL_i.

1.3 The implementation in MPICH's performance test suite

I implemented that new methodology to time `MPI_Bcast` accurately as a plug-in module of the performance test suite "perftest" distributed with MPICH. The code was written in C because the MPI Standard defines only Fortran and C bindings and the rest of "perftest" is written in C. The code I wrote is available at
`http://www-unix.mcs.anl.gov/~lacour/argonne2001/perftest`.

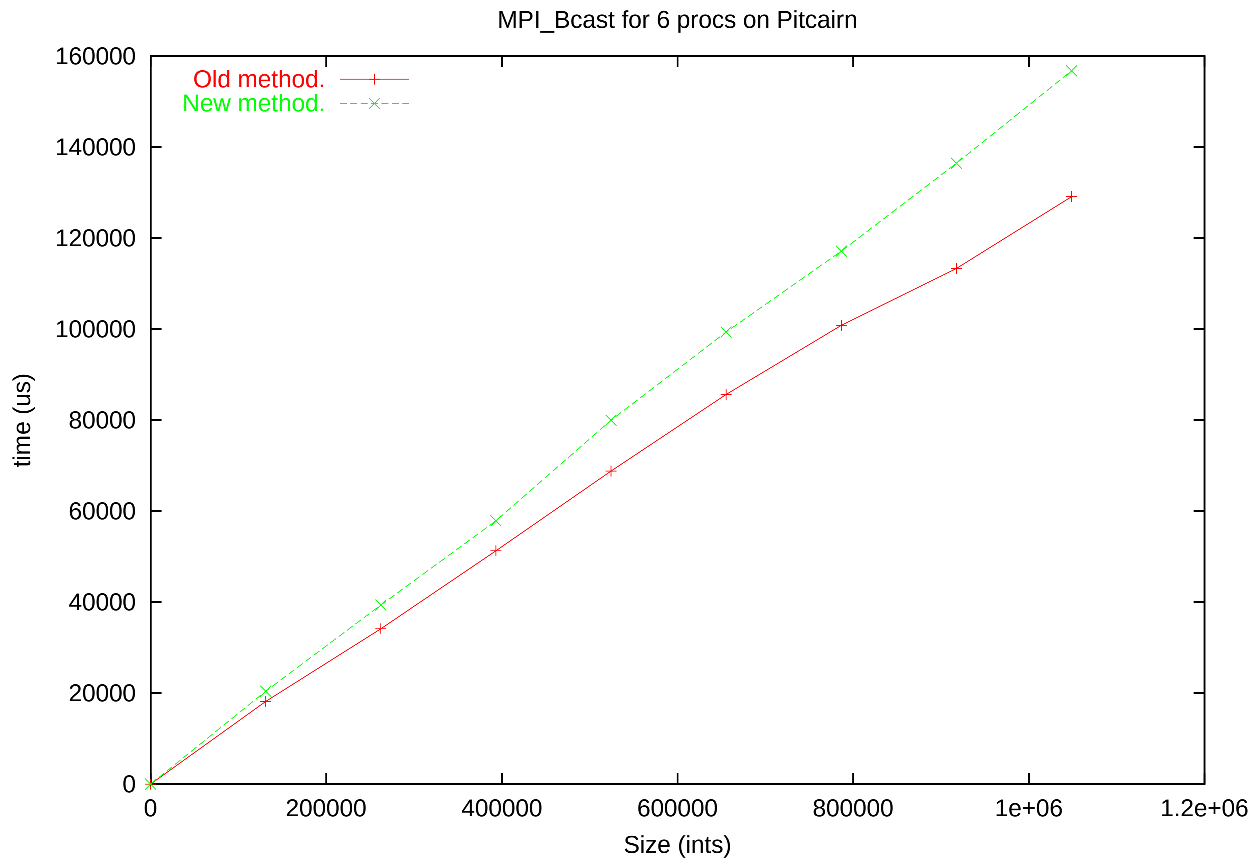


Figure 8: Results of the new timing methodology compared with the previous simple algorithm's results. `MPI_Bcast` is performed for various message sizes with 6 processes.

This implementation was tested and a few graphs were plotted (see figure 8 and the web site mentioned above): as expected they show that `MPI_Bcast` takes more time when it is evaluated with the new methodology because it does not take advantage of the pipelining effect.

2 MPICH-G2 aware of the underlying topology

2.1 Topology of processes and protocol levels

The Globus-2 device of MPICH can be used to run MPI programs in "computational grids". These environments are highly heterogeneous: they can be made of several LANs (Local-Area Networks) connected over a WAN (Wide-Area Network) using the TCP protocol; each LAN can be composed of different machines communicating via TCP over the LAN. A given machine can be *vendor-MPI* equipped: this is an efficient MPI library provided by the vendor (SGI, IBM, Sun, ...) allowing fast MPI communications between processes running on the same host. Another machine can have several CPUs exchanging messages via shared memory. The IBM SP can be equipped with a high-performance switch to allow its nodes to communicate quickly, a cluster of PCs can have its nodes connected with Myrinet, ... All these communication protocols have very different bandwidths and latencies. They can be sorted as in table 1 in function of their latencies.

level 0	>	level 1	>	level 2	>	level 3, 4, ...
WAN-TCP	>	LAN-TCP	>	localhost-TCP	>	shared memory vendor MPI MyriNet

Table 1: Sorted protocols in function of their latencies.

MPICH-G2 assumes that the TCP protocol is present on all machines and every host can communicate directly with all the other machines. Typically, the TCP latency over a WAN is ~ 200 times the latency over a LAN. A TCP communication between two processes running on the same machine is faster than between two different machines in the same LAN because the TCP packets do not go onto the wire: the network card is bypassed thanks to an optimization of the TCP protocol.

A realistic example of configuration is given on figure 9: Argonne Nat'l Lab. (ANL, Chicago) and Lawrence Livermore Nat'l Lab. (LLNL, California) are connected over a WAN. *Baby* and *Blue* are 2 IBM SPs equipped with 4-way SMP nodes connected over IBM's high-performance switch. *Denali* is a 96-CPU Origin 2000 with SGI's implementation of MPI installed and *Pitcairn* is an 8-CPU Sun Enterprise without vendor-MPI (its only communication protocol is TCP). Figure 9 shows that the processes on *Denali* will never use the level-2 protocol (localhost-TCP) since they have a more efficient way of communicating with SGI's MPI.

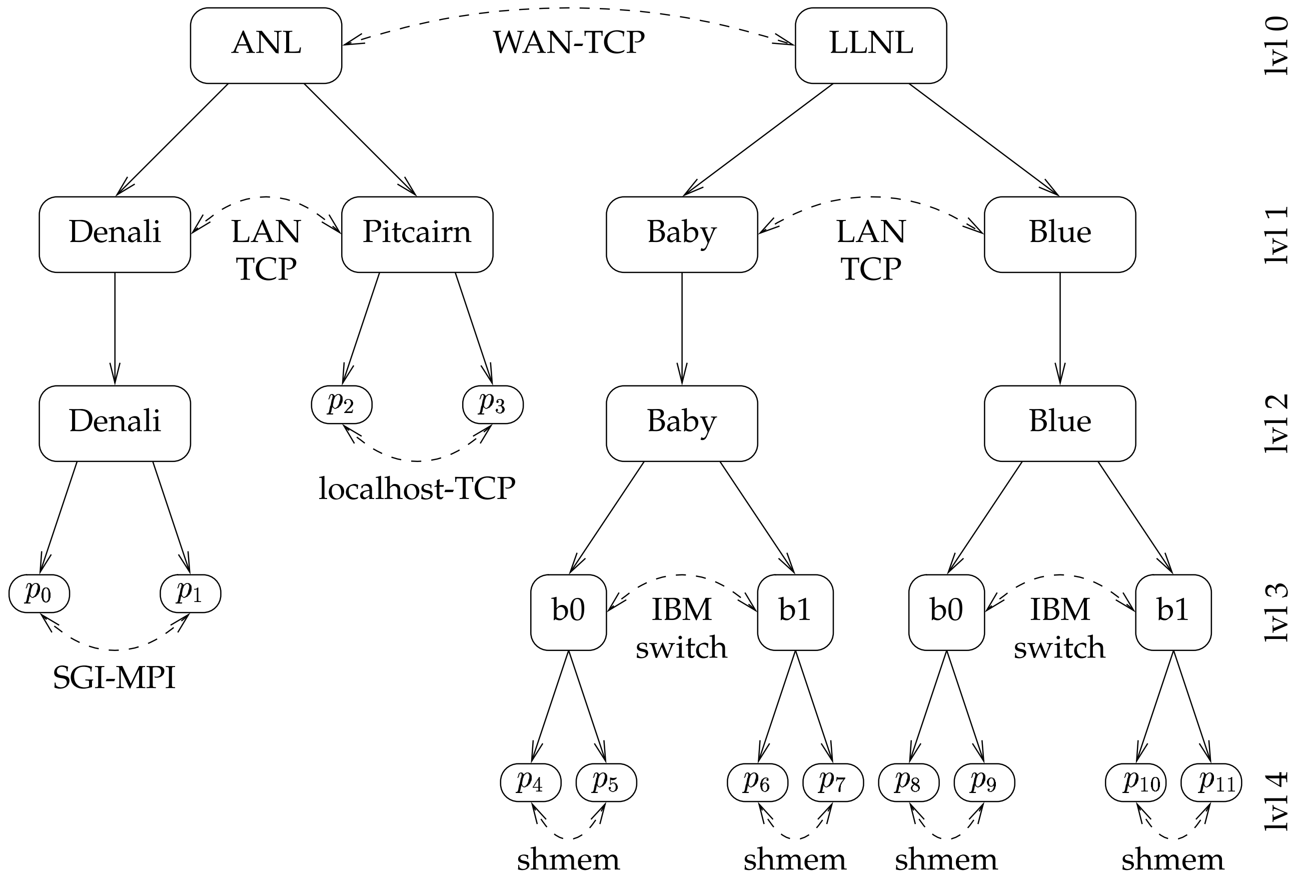


Figure 9: An example of realistic configuration.

The WAN level, LAN level and vendor-MPI level had already been conceived. I suggested the introduction of the localhost-TCP level and I wrote the code for this addition.

2.2 Colors and clusters

The Globus-2 device provides an access to the channels of each process. A channel can be seen as the list of protocols a process can speak, sorted from the fastest (in terms of latencies) to the slowest protocol (TCP). As defined in [5], a cluster at level i is a set of processes which can talk together using their protocol of level i. In the example of figure 9, the clusters are those shown in table 2.

At each level, all the processes which belong to the same cluster are affected the same color. At a given level, each cluster has a unique color. Table 3 shows the colors of each process at every level.

level	clusters			
0	$\{ p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11} \}$			
1	$\{ p_0, p_1, p_2, p_3 \}$		$\{ p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11} \}$	
2	$\{ p_0, p_1 \}$	$\{ p_2, p_3 \}$	$\{ p_4, p_5, p_6, p_7 \}$	$\{ p_8, p_9, p_{10}, p_{11} \}$
3	$\{ p_0, p_1 \}$		$\{ p_4, p_5, p_6, p_7 \}$	$\{ p_8, p_9, p_{10}, p_{11} \}$
4			$\{ p_4, p_5 \}$ $\{ p_6, p_7 \}$	$\{ p_8, p_9 \}$ $\{ p_{10}, p_{11} \}$

Table 2: The clusters of processes in the particular example.

level	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	3	3	3	3
3	0	0			1	1	1	1	2	2	2	2
4					0	0	1	1	2	2	3	3

Table 3: The colors of the clusters and processes.

Inside each cluster at level i, the processes are enumerated and assigned a unique cluster identifier (cluster ID), unless they might have already been enumerated at any deeper level. Table 4 shows the cluster IDs for the particular configuration of figure 9.

level	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	1	1	0	0	0	0	1	1	1	1
2	0	0	0	1	0	0	0	0	0	0	0	0
3	0	1			0	0	1	1	0	0	1	1
4					0	1	0	1	0	1	0	1

Table 4: The cluster IDs of the processes at each protocol level.

2.3 Automatic computation of the colors and cluster IDs

The Globus system provides information about the channels. A code was already written by Nick Karonis to generate the tables of colors and cluster IDs given the channels (sorted protocol lists). I corrected and tested that implementation and I integrated it into the Globus-2 device of MPICH. Both tables (colors and cluster IDs) are equivalent: they provide the same information on the topology. However I chose to keep both of them in memory because both representations of the topology will be directly used later (see section 2.5): the aim of the Globus-2 device is high-performance, keeping both tables in memory avoids needless re-computations.

This information is computed dynamically each time a communicator[5] is created and both tables are included into the communicator's data structure. Of course, they are destroyed (and memory freed) as the communicator vanishes.

2.4 The automatic determination of the LAN

The LAN which a process belongs to is the only value the Globus system does not provide.

- I envisaged an automatic determination of the LAN counting the number of "*hops*" (routers) separating every processors, using the TTL field ("*time to live*") of the TCP headers, just like the UNIX command `traceroute`. However that requires to use "*raw* TCP sockets" demanding root's priviledges: MPICH is a user-level library so the idea was abandonned.

- Evaluating the TCP-latency between two processes could also be used to determine whether both processes are in the same LAN or not (using a `ping`-like method), but that is not reliable enough and some firewalls would filter that kind of packets. Moreover it could take a very long time to evaluate the latency for every couple of processes.

- Another idea consists in comparing the IP addresses of the processes to guess if they are on the same LAN or not. However that method is not reliable since we can perfectly imagine that two machines are in the same *logical* IP domain but located on different continents.

Finally, we decided to rely on the user to specify (optionally) the LAN through an environment variable.

2.5 The sets of communicating processes

In the example of figure 9, if process p_0 broadcasts a message (`MPI_Bcast`, root = p_0), we would like to see the following scheme:

1. p_0 sends the message to p_4 using the slowest protocol (WAN-TCP),

2. p_0 sends to p_2, p_4 sends to p_8,

3. p_0 sends to p_1, p_2 sends to p_3, p_4 sends to p_6, p_8 sends to p_{10},

4. p_4 sends to p_5, p_6 sends to p_7, p_8 sends to p_9, p_{10} sends to p_{11}.

level	sets of communicating processes
0	$\{ p_0, p_4 \}$
1	$\{ p_0, p_2 \}$ $\{ p_4, p_8 \}$
2	$\{ p_0 \}$ $\{ p_2, p_3 \}$ $\{ p_4 \}$ $\{ p_8 \}$
3	$\{ p_0, p_1 \}$ $\{ p_4, p_6 \}$ $\{ p_8, p_{10} \}$
4	$\{ p_4, p_5 \}$ $\{ p_6, p_7 \}$ $\{ p_8, p_9 \}$ $\{ p_{10}, p_{11} \}$

Table 5: Sets of communicating processes at each level.

That scheme *minimizes the communications through the slowest protocols* (high-latency communications) and executes them first. In that particular case, the "*sets of communicating processes*" are those shown in table 5.

I implemented the generation of the sets of communicating processes using the color and cluster ID tables. These sets are needed by every optimized topology aware MPI collective operation (see section 3). These sets allow each process to guess from what process it will receive a message and to what other process it will have to relay the message. The important point is that the generation of these sets requires ***no** communications between the processes*: that generation is very fast because every process computes the sets in which it will be involved *locally*.

To make it easier to compute the sets of communicating processes from the cluster ID table, I had the idea to rename the clusters such that the root process of an asymetric collective operation[6] have only zeros as cluster IDs at each level. Inside each cluster at every level, there is a *master process* (the "representative" of the cluster at the given level): this particular process has cluster ID = 0 at any level greater than or equal to the given level. In the particular configuration taken as an example (table 4), process p_0 is a master process at every level (it might be the root of an asymetric collective operation), process p_1 is never a master, process p_8 is the representative (master) of its clusters at levels 4, 3, 2. At level i, a set of communicating processes is made of the master processes of the clusters at level $i + 1$.

The code[7] I wrote was fully (and successfully) tested on both simulated and real configurations. I paid attention to write an optimized code, helping the compiler (expliciting the constants, defining the variables as locally as possible, ... to produce high-performance executable files). All the return values of the functions called are always checked. The code was written in C because the rest of the software (MPICH) is in C[8] and also because we need high performance and have good C-compilers.

[5]Roughly speaking, a communicator is a set of processes in MPI terminology. For more details, see [4].

[6]The "asymetric" collective operations are those with a particular root process specified: `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Reduce`, ...

[7]See my web page: `http://www-unix.mcs.anl.gov/~lacour/argonne2001/topology`.

[8]Serious people doing some software engineering don't write their programs neither in Prolog nor in Lisp.

3 Topology aware collective operations

A point-to-point communication between two processes can occur if and only if they both belong to the same "set of communicating processes" (as defined in section 2.5). Each process executing a topology aware MPI collective operation starts with the communications inside the set of communicating processes at level 0 (highest latency) and finishes with the set at the deepest level (fastest protocol). Inside a set of communicating processes at a given level (*i.e.*: using a given protocol), the communication pattern depends on the following parameters:

- the message size,
- the number of processes (p) involved in the set,
- the communication latency (L) and bandwidth,
- the send/receive overheads $(o_s \simeq o_r) = o$.

The following MPI functions were implemented in a topology aware manner and successfully tested (MPICH-G2 passed the MPICH test suite). The source codes are available at `http://www-unix.mcs.anl.gov/~lacour/argonne2001/taco`.

3.1 Topology aware `MPI_Barrier`

The flat-tree algorithm (see figure 10) for `MPI_Barrier` works as follows: each process of the set sends an empty message to all the processes of the set and receives an empty message from all the processes too: this pattern can lead to network congestion if there are too many processes involved in the set. However it is used at the WAN-TCP protocol level because the users of MPICH-G2 usually run their applications over few LANs and the number of processes which communicate at the WAN-TCP level are not numerous. As illustrated on figure 11, the time it takes to perform `MPI_Barrier` using the flat-tree algorithm is (p is the number of processes in the set):

$$\begin{cases} t = 2(p-1)o & \text{if } L \leq (p-2)o, \\ t = po + L & \text{if } L \geq (p-2)o. \end{cases}$$

The hypercube algorithm is more scalable and used for all the other protocol levels. In this pattern (see figure 12), each process notifies its $log_2 p$ neighbors in the hypercube that it has reached the barrier and waits for its $log_2 p$ neighbors to reach the barrier too. Then each process sends a "GO" signal to its $log_2 p$ neighbors and waits for the "GO" signal from its $log_2 p$ neighbors. In case the number of processes in the set is not a power of 2, one just needs to add some virtual processes to reach the next power of 2. The time it takes to perform `MPI_Barrier` using the hypercube algorithm is (see figure 13):

$$\begin{cases} t = 2o\lceil log_2 p\rceil & \text{if } 2L + 3o \leq o\lceil log_2 p\rceil, \\ t = 2L + (3 + \lceil log_2 p\rceil)o & \text{if } 2L + 3o \geq o\lceil log_2 p\rceil. \end{cases}$$

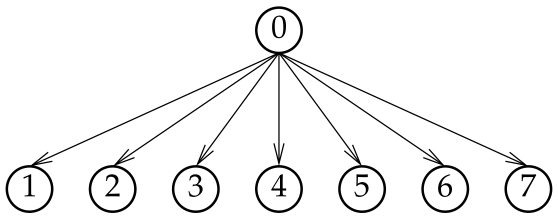


Figure 10: A flat tree.

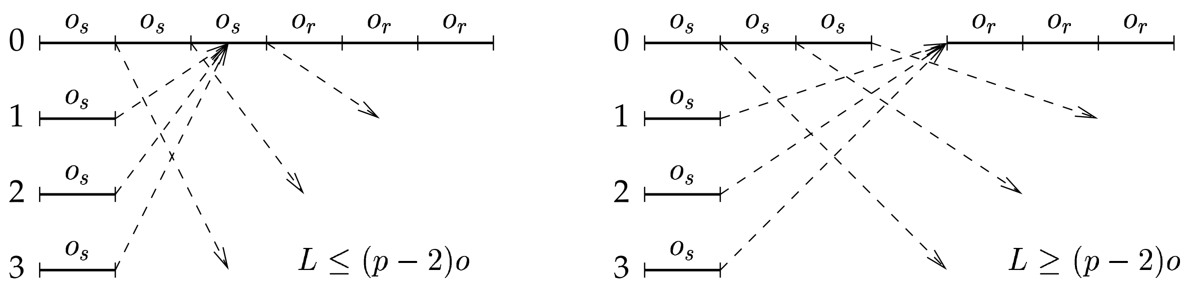


Figure 11: Flat-tree `MPI_Barrier` for $p = 4$ at process 0.

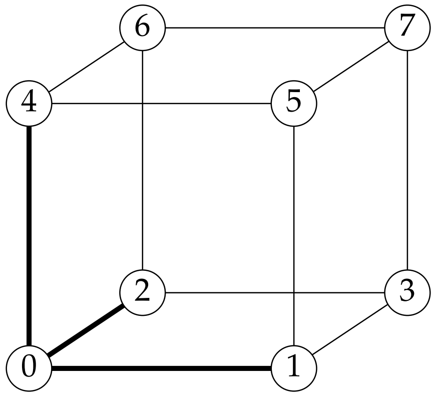
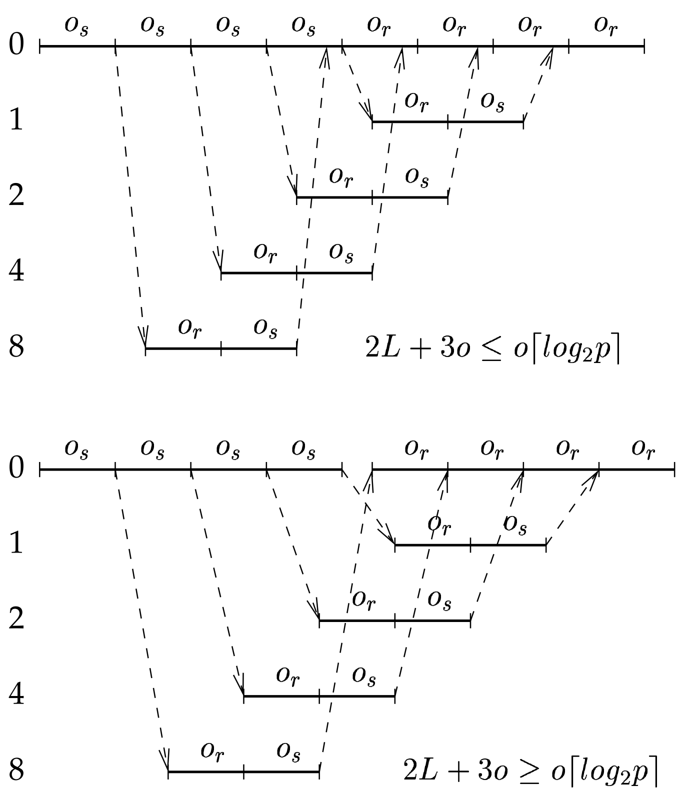


Figure 12: Process 0 communicating with its 3 neighbors in a hypercube of dimension 3.

Figure 14 shows it is more interesting to use the flat-tree algorithm to perform `MPI_Barrier` when the latency is high (WAN-TCP). The number of LANs interconnected cannot reasonably be very high, so the number of processes communicating at the WAN-TCP protocol level is small. Figure 15 shows it is more efficient to use the hypercube algorithm when the latency is small (fast protocols): the users of MPICH-G2 generally run their MPI applications on a few large vendor MPI equipped machines with many processes running on a single machine.



o_s
o_r
0
1
2
4
8
$2L + 3o \leq o\lceil log_2 p \rceil$
$2L + 3o \geq o\lceil log_2 p \rceil$

Figure 13: Hypercube `MPI_Barrier` for $p = 16$ at process 0.

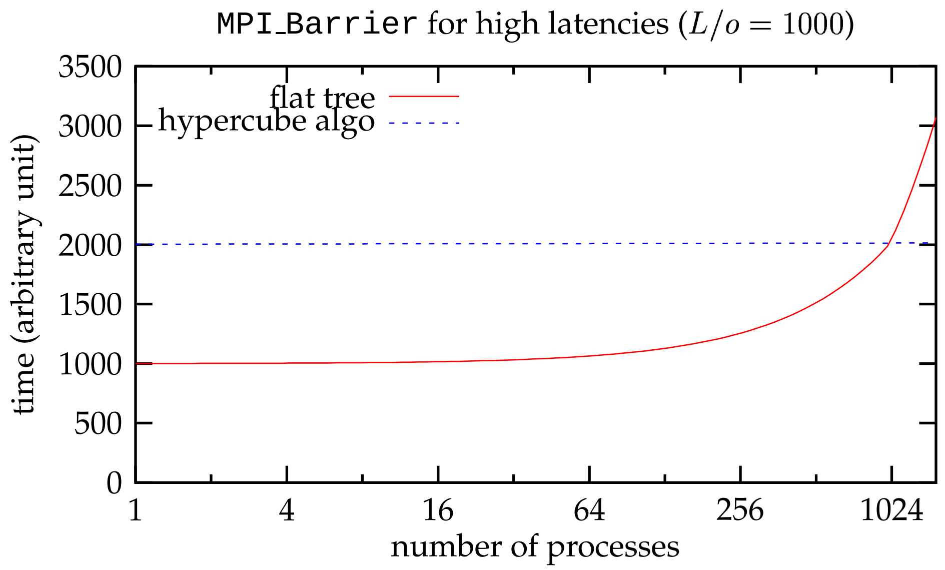


Figure 14: Simulated comparison between the flat-tree and the hypercube algorithms for `MPI_Barrier` with a high latency.

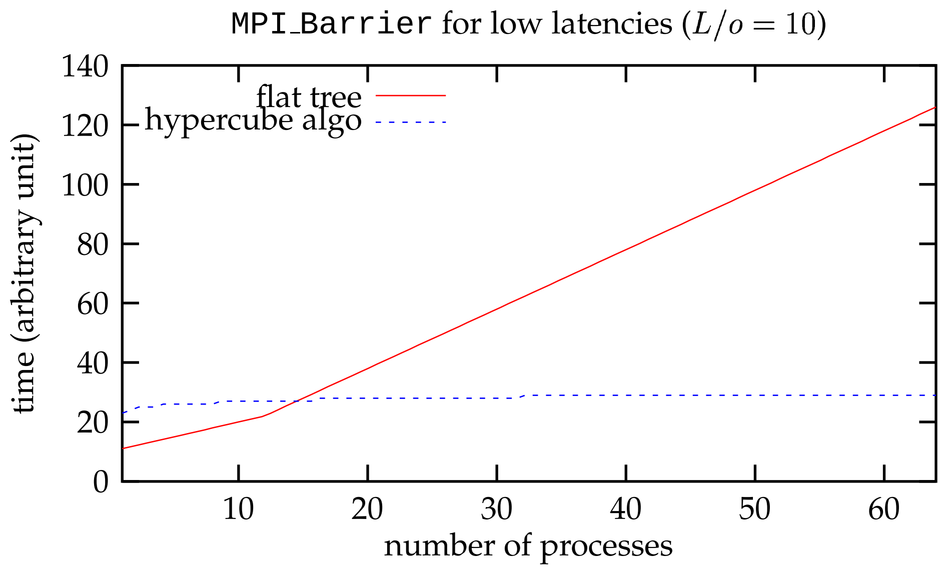


Figure 15: Simulated comparison between the flat-tree and the hypercube algorithms for `MPI_Barrier` with a low latency.

3.2 Topology aware `MPI_Bcast`

In the flat-tree algorithm, the root process sends the message to each process in the set. That might result in network congestion if there are too many processes involved. The time it takes to perform a flat-tree `MPI_Bcast` is: $t = po + L$ (see figure 16).

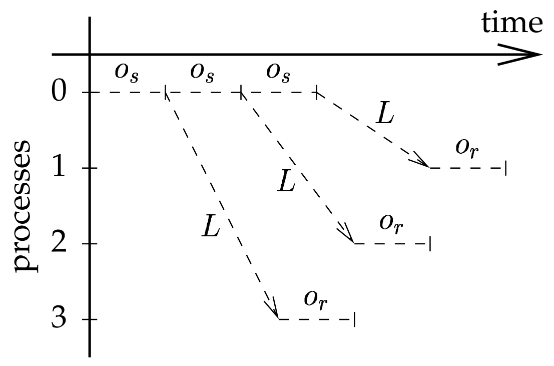


Figure 16: An example of flat-tree `MPI_Bcast` with root = 0 and p = 4.

A binomial tree of degree 0 (b_0) is just one node. A binomial tree of degree n (b_n) is a node with n sons which are n binomial trees of degrees b_0, b_1, b_2, ..., b_{n-1}. Thus a binomial tree of degree n is made of 2^n nodes. Figure 17 shows a binomial tree of degree 4 (16 nodes).

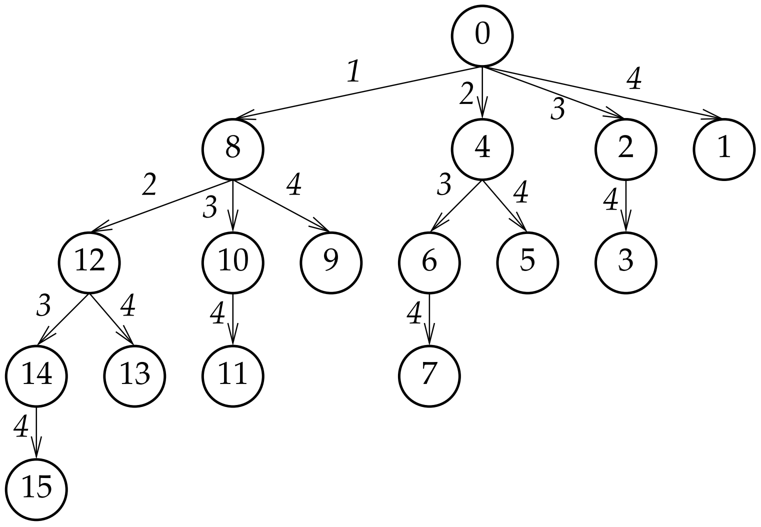


Figure 17: A binomial tree of degree 4 (16 nodes).

Using a binomial-tree algorithm, if process 0 is the root of an `MPI_Bcast` in a set of 8 processes, then the communication pattern is the one shown on figure 6. To perform a binomial-tree `MPI_Bcast`, $t = (2o + L)\lceil log_2 p\rceil$.

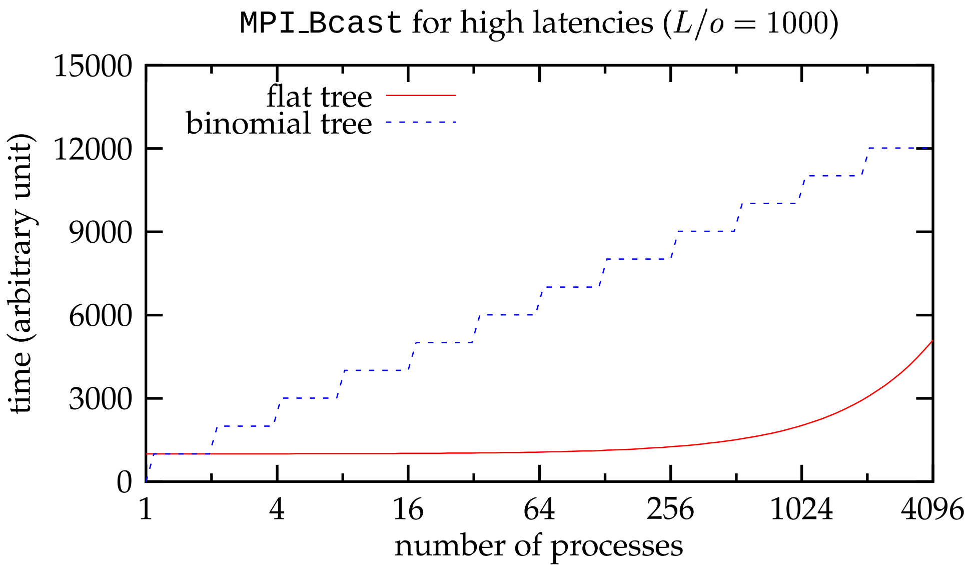


Figure 18: Simulated comparison between the flat-tree and the binomial-tree algorithms for `MPI_Bcast` with a high latency.

As illustrated on figures 18 and 19, I chose the flat-tree algorithm to broadcast at protocol level 0 (WAN-TCP, high latency, few processes) and the more scalable binomial-tree algorithm at all the other levels (small latencies and possibly large number of processes).

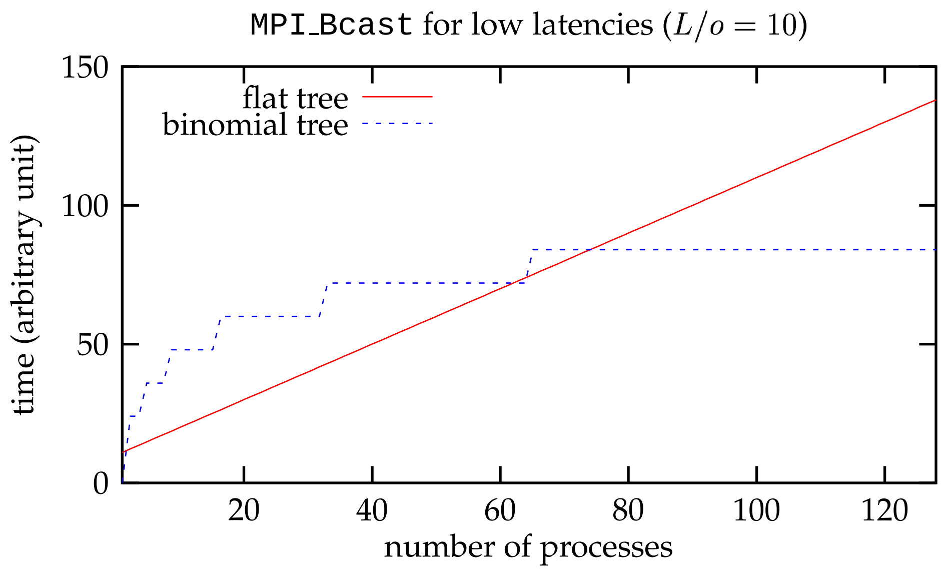


Figure 19: Simulated comparison between the flat-tree and the binomial-tree algorithms for `MPI_Bcast` with a low latency.

3.3 Topology aware `MPI_Gather` and `MPI_Scatter`

As for `MPI_Bcast`, `MPI_Gather` and `MPI_Scatter`[9] are implemented using a flat-tree algorithm at the WAN-TCP protocol level and a binomial-tree algorithm at any other level. For `MPI_Scatter`, the trees are traversed from the top (the root) down to the bottom (like `MPI_Bcast`), while they are traversed from the bottom up to the top for `MPI_Gather`.

The two major problems were essentially technical and are not detailed here: they concerned non-contiguous MPI data types and also when the number & type of the items sent do not match those of the items received (which is allowed by the MPI Standard). Another difficulty was to allocate enough memory for each process to store temporarily some data items which would be relayed to other processes. For instance, in an `MPI_Scatter` from root = p_0 to 12 other processes using a binomial-tree algorithm (see figure 17), process p_8 will receive from p_0 the data items for p_8, p_9, p_{10}, p_{11}, p_{12}; so p_8 must allocate memory to store the data to relay to p_9, p_{10}, p_{11}, p_{12}, and p_8 must also be aware it must send to p_{10} the data destined to p_{11}.

[9]See the semantics of the collective operations in [4].

3.4 Topology aware `MPI_Reduce`

`MPI_Reduce`[10] performs a mathematical operation on all the data items provided by all the processes in the MPI communicator and stores the result at the root process. The operation is supposed to be associative, but *not compulsorily commutative*. Let x_i be the data item provided by process p_i, then the result of `MPI_Reduce` for the operation $\star$ must be: $x_0 \star x_1 \star x_2 \star \cdots \star x_i \star \ldots$.

Commutative operations

When the operation $\star$ is commutative, the computation order does not matter. Once again, I implemented `MPI_Reduce` using a flat-tree algorithm at protocol level 0 (WAN-TCP) and a binomial-tree algorithm at any other level (low latency protocols). For example, an `MPI_Reduce` using a binomial-tree algorithm to root = p_0 with the commutative operation $\star$ in a set of 7 processes is made of the following steps (refer to figure 17):

1. p_5 sends x_5 to p_4, p_3 sends x_3 to p_2, p_1 sends x_1 to p_0;

2. p_4 computes $x_{45} = x_4 \star x_5$, p_2 computes $x_{23} = x_2 \star x_3$, p_0 computes $x_{01} = x_0 \star x_1$;

3. p_6 sends x_6 to p_4, p_2 sends x_{23} to p_0;

4. p_4 computes $x_{456} = x_{45} \star x_6$, p_0 computes $x_{0123} = x_{01} \star x_{23}$;

5. p_4 sends x_{456} to p_0;

6. p_0 computes the final result $x = x_{0123} \star x_{456} = ((x_0 \star x_1) \star (x_2 \star x_3)) \star ((x_4 \star x_5) \star x_6)$.

Non-commutative operations

Using the sets of communicating processes to perform an `MPI_Bcast` may give a different result from the one imposed by the MPI Standard because the topology information imposes its own computation order, depending on the root process for instance. There were two possibilities:

1. Either ignoring the topology and protocol levels: that solution may incur several WAN-TCP latencies in sequence, hence very bad performance results;

2. Or gathering all the data items to the root using the optimized topology aware `MPI_Gather` I wrote, and leaving the root process compute all the $(p-1)$ operations (instead of distributing the computation tasks to the nodes as in the commutative case).

[10]For more details on `MPI_Reduce`, see [4].

I chose to implement the second solution because incurring several WAN-TCP latencies ($\sim$ 200 ms each) takes much longer than computing $(p - 1)$ operations on one node (even for matrix-matrix multiplication). That is true if the message size is below a certain threshold; this threshold reflects the tradeoff between:

- *distributed* computations and several WAN-TCP latencies in sequence,
- *centralized* computations and only one WAN-TCP latency.

The threshold cannot be calculated exactly because it depends on both the WAN-TCP latency and the speed & load of the CPU of the root process, but it can be estimated to be higher than the usual message sizes (large matrices). For an order of magnitude, during a typical WAN-TCP latency ($\sim$ 200 ms), a 1 GHz CPU can perform 200 million elementary assembly operations; if there are 100 $\star$ operations to perform (that is an `MPI_Reduce` over 101 processes working in parallel), a single node can do 2 million elementary operations during one WAN-TCP latency.

Making this particular choice, I also chose not to comply with an *advice to implementors* taken from [8], but the implementation remains fully compliant with the MPI Standard requirements:

> *It is strongly recommended that `MPI_Reduce` be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of the processors.*

In my implementation, depending on the root process, the computation may occur with different associativities and on different architectures (with different round-off modes, floating-point number representations...), hence possibly slightly different results.

3.5 Future work

Time did not allow me to write a topology aware implementation of the nine other MPI collective operations[11]. However I re-wrote the most widely used collective operations and all the remaining functions look like the ones I implemented:

- `MPI_Gatherv` can be made topology aware imitating `MPI_Gather`,
- `MPI_Scatterv` looks like `MPI_Scatter`,
- `MPI_Scan` is inspired from `MPI_Reduce`.

A performance evaluation of the new implementation of the collective operations could also be carried out to confirm the improvements.

[11] `MPI_Gatherv`, `MPI_Scatterv`, `MPI_Allgather`, `MPI_Allgatherv`, `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Allreduce`, `MPI_Reduce_scatter`, `MPI_Scan`.

4 The underlying topology now available at the user level

Some users of MPICH-G2 asked for a way to access the physical topology of the processes from their applications. That could be useful for instance to create new communicators[12] with `MPI_Comm_split` such that all the processes in each resulting communicator be in the same LAN (Local-Area Network) for faster communications.

I implemented that feature in MPICH-G2 using the attribute caching facility attached to each communicator. Thanks to two global "keys" added to the `"mpi.h"` include file, the user now has an access to the number of protocol levels of each process in the communicator as well as to the "colors" (as defined in section 2.2) of the processes.

I also wrote two MPI example programs.

- A program to test that new feature and the robustness of the implementation: for instance, the user does not have a direct access to the pointers to the arrays used by the MPICH-G2 library to prevent him from modifying the internal data. The user has only an access to a copy of the arrays.
- Another program to show the user of MPICH-G2 how to take advantage of the access to the underlying topology in order to create a new communicator for each LAN (allowing fast communications inside each new communicator).

All the source codes can be seen at:
`http://www-unix.mcs.anl.gov/~lacour/argonne2001/user_access`.

[12]Roughly speaking, a communicator is a set of processes in MPI terminology. For more details, see [4].

Acknowledgements

I am grateful to Ian Foster for accepting me in his research group at MCS. I thank Nick Karonis for trusting me and letting me modify some parts of the Globus-2 device, as well as for his precious explanations about MPICH-G2. I also thank Bill Gropp for his patience and for allowing me to modify some central files of MPICH.

References

[1] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, 1993.

[2] B. de Supinski and N. Karonis. Accurately Measuring MPI Broadcasts in a Computational Grid. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1999.

[3] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *SuperComputing Conference*. ACM Press, 1998.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, second edition, 1999.

[5] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *Fourteenth International Parallel and Distributed Processing Symposium*, pages 377–384, May 2000.

[6] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999.

[7] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. MPI's Reduction Operations in Clustered Wide Area Systems. In *Message Passing Interface Developer's and User's Conference (MPIDC'99)*, pages 43–52, Atlanta, GA, March 1999.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, March 1994. `http://www.mpi-forum.org/docs/mpi-11.ps`.

[9] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast Collective Communication Libraries, Please. In *Proceedings of the Intel Supercomputing Users' Group Meeting*, June 1995.